Strings, characters and character literals

Internally, computers only manipulate bits of data; every item of data input can be represented as a number encoded in base 2. However, when it comes to processing text, mathematical equations on the numeric value of letters and punctuation are not very comfortable to write and use.

To optimize the manipulation of text, many programming language provide extra facilities that help the programmer.

The two main constructs available are:

- a simple data type to *represent a single character* - a letter, digit, punctuation sign, etc.
- a more complex data type to *represent a sequence of zero or more characters*. This is called a *character string*.

Character types

Character types are not strictly necessary – for example, Python does not have a dedicated character type, and programmers will use variables of type `int` to manipulate individual characters instead.

However, *dedicated character types are useful because they are more compact in memory*: they occupy typically much less space (1-2 bytes) than an ordinary integer (4-8 bytes). So if a program manipulates many characters it is more memory efficient to use the character type when it exists. For example, Java even has two character types, called `char` and `byte`, much smaller than `int` that is 32-bit long:

Data type	**Size**	**Set of allowable values**	**Adequate for**
`char`	16 bits	0 to 65535	Processing international text
`byte`	8 bits	-128 to 127	Processing binary files

Which type to use depends on the application: if text is to be read in or printed for human users, usually `char` is adequate; whereas reading or writing character for use by another program should probably use `byte` instead.

String types and Java's `String`

Every language has a different way to manipulate sequences of zero or more characters. Some languages like C do not have a dedicated type for character strings, so a programmer has to "build their own" using arrays of individual characters. In Java, like in Python and most modern languages, there is a dedicated type; in Java, the name for this is `String`.

Like numbers, strings can be read in (`Scanner.next()`) and printed out (`PrintStream.println()`, `.printf()`) as-is. We can also express a string literal in the Java program using double quotes:

```
String a = "hello";
```

This frament creates a variable of type `String` and initializes it to store the sequence of characters 'h', 'e', 'l', 'l', 'o'.

Testing for `String` equality

Java's direct equality comparison operators "`==`" and "`!=`" are *only defined for primitive types*. This is why we can compare two `int`'s using "`==`" (`int` is a primitive type), however *the direct comparison operators do not work on compound types* like `String`.

This is arguably a shortcoming of the Java language in particular: most other programming languages do allow programmers to use `==` to compare character strings. So unfortunately you will have to remembed this as an exception. It is especially unfortunate because *Java does not inform you properly if you make the mistake to use == to compare strings*. For example, with the following code:

```
String a = "abc";
String b = "abc";
if (a == b)
    out.println("all is well");
```

the program will not print "all is well" as expected, because "`==`" silently did something else entirely.

The full explanation of what is going on here will come only later (when talking about references and aliasing), so at this point **just remember this:** *to compare strings, use the method equals()*, like this:

```
String a = "abc";
String b = "abc";
if (a.equals(b))
    out.println("all is well");
```

This program fragment properly prints "all is well" as expected.

String and number conversions

Strings are not numbers, so we cannot perform numeric arithmetic on them:

```
String b = "123";
String c = b / 4; // produces an error: b is not a numeric variable
```

The only "arithmetic" operation available with Java's `String` is "`+`", which concatenates (attaches together) two strings:

```
String b = "123";
String c = b + b;
out.println(c); // prints 123123
```

To convert a `String` to a number (`int`, `double`, etc.) you can use the `Scanner`, already seen previously. To perform the opposite conversion (from `int`, `double`, etc. to `String`), `String` provides a service called `valueOf`:

```
String d = String.valueOf(3.1415); // d = "3.1415"
String e = d + d;
out.println(e); // prints 3.14153.1415
```

Useful `String` methods

The type `String` provides numerous services that help with the use of character strings in programs. A complete list is provided in the online documentation of the Java language, so you look up `String` there every time you are looking for a string-related language feature.

However it may be useful to "pre-load" the following knowledge in your long-term memory:

- `a.equals(b)` compares two strings `a` and `b` for equality (replaces "`==`");
- `a.compareTo(b)` compares two strings `a` and `b` for alphabetical order (replaces "`<`" and "`>`")
- `a.length()` returns the length of the string `a`, ie the number of character it contains;
- `a.charAt(i)` retrieves the character in the string `a` at position `i`;
- `a.substring(i, j)` returns the part of the string that begins at position `i` and ends at position `j`, as a new string;
- `a.indexOf(c)` returns the position of the first occurrence of the character `c` in `a`.

Also even if you do not need to remember the specifics by heart, you should remember that `String` provides other services that help with handling international text, in particular comparison while ignoring the difference between lower and upper case, case conversion, etc.

Literals, character names and escape sequences

A literal is a construct in a programming language whose value is equal to what it represents. For example, the text "123" formed by 3 digits '1', '2', '3' concatenated together is a valid integer literal, whose value is equal to a hundred and twenty three.

In every programming language, *there are some values which can be represented by multiple different literals.* For example, The approximate value represented by the literal "`3.1415`" is also represented by "`31.415e-1`" ($31.415 \times 10^{-1} = 3.1415$).

Characters in particular typically have many names in programming languages. For example, the following 3 literals all encode the same value in Java:

```
'a'
'\x61'
'\u0061'
'\141'
```

The first form is the most simple, this this the character itself.

The other three forms are other encoding that start with a backslash ("`\`") followed by a code. The presence of the backslash indicates that what follows must be interpreted differently. A construct of this form is called a *character escape sequence*.

The reason why these special forms with a backslash are useful is that some characters cannot be easily entered in a text editor. For example, what if I wanted to check whether a string starts with the Greek character alpha? In most environments I cannot type it in the source code because special characters are not allowed in the text editor, or not saved properly. In order to work around this shortcoming, I can use the character code instead:

```
final char GREEK_ALPHA = '\u0251';

if (a.charAt(0) == GREEK_ALPHA) ...
```

Java, like C, Python and many other programming languages recognizes the following escape sequences:

Sequence	Meaning	Example	Same as (examples)
`\uNNNN`	Unicode character with code NNNN	`'\u0061'`	`'a'`
`\xNN`	Character with hexadecimal code NN	`'\x61'`	`'a'`
`\NNN`	Character with octal code NNN	`'\141'`	`'a'`
`\'`	The apostrophe character		`'\x27','\047','\u0027'`
`\"`	The double quote character		`'\x22','\042','\u0022'`
`\\`	The character "`\`" itself		`'\x5c','\134','\u005c'`
`\n`	The newline character		`'\x0a','\012','\u000a'`
`\t`	The tab character		`'\x09','\011','\u0009'`
`\r`	The carriage return character		`'\x0d','\015','\u000d'`
`\f`	The form feed (new page) character		`'\x0c','\014','\u000c'`

To summarize, *if a character can be entered as-is in the program text, use that directly; otherwise, use a character escape sequence.*

Important concepts

- *character type* and *string type* and the difference between them;
- dedicated character types are *more compact in memory* than other integer types;
- testing for equality using `equals()`, do not use "`==`";
- conversion between `String` and numbers in Java using `Scanner` and `String.valueOf()`;
- `String`'s `charAt`, `compareTo`, `length`, `substring` and `indexOf` methods;
- *character literals*;
- *character escape sequences*.

Further reading

- Think Java, sections 8.1-8.6 (pp. 91-96)
- Introduction to Programming, sections 2.2.3-2.2.4 (pp. 26-28), section 2.3.3 (pp. 34-35)
- Absolute Java, section 1.3 (pp. 33-45)

- WikiBooks: Programmeren in Java, Stringbewerkingen

Copyright and licensing